Acta Informatica (2019) 56:391–404
https://doi.org/10.1007/s00236-019-00335-9

ORIGINAL ARTICLE

Red–black trees with constant update time

Amr Elmasry[1] · Mostafa Kahla[2] · Fady Ahdy[2] · Mahmoud Hashem[2]

Received: 21 December 2017 / Accepted: 8 May 2019 / Published online: 21 May 2019

Abstract
We show how a few modifications to the red–black trees allow for constant worst-case update time (once the position of the element to be inserted or deleted is known). The resulting structure is based on relaxing some of the properties of the red–black trees while guaranteeing that the height remains logarithmic with respect to the number of nodes. Compared to the other search trees with constant worst-case update time, our tree is the first to provide a tailored deletion procedure without using the global rebuilding technique. In addition, our procedures are simpler and allow for an easier proof of correctness than those alternative trees.

1 Introduction

Balanced search trees are among the most common and well-studied data structures used in various algorithms. They provide an elegant solution to the dictionary problem, in which one needs to maintain a dynamic set of elements. In the comparison-based model, searching, inserting, and deleting an element are carried out in $O(\log n)$ time, where n is the number of elements in the tree. There exists a great variety of balanced search trees in the literature. Examples include: AVL trees [1], B-trees [3,4], red–black trees [9], height-balanced trees [11], and weight-balanced trees [16].

Guibas and Sedgewick [9] introduced the red–black trees as a fundamental balanced search tree that was derived from the symmetric binary B-trees (popular as 2–4 trees) [3]. Tarjan [23] gave a version of red–black trees that performs a constant number of structural changes (rotations) per operation. Still, the worst-case rebalancing time for Tarjan's trees is logarithmic. Andersson [2] simplified the insertion and deletion operations for red–black trees, simulating 2–3 trees rather than 2–4 trees. Sedgewick [22] gave a variation that he calls left-leaning red–black trees. Okasaki [19] showed how to make the insertion operation for red–black trees purely functional.

Normally, any insertion or deletion in balanced search trees comprises three steps:

1. querying the position of the key; this step takes $O(\log n)$ time

✉ Amr Elmasry
elmasry@alexu.edu.eg

[1] Department of Computer Engineering and Systems, Alexandria University, Alexandria, Egypt

[2] Computer and Communications Engineering Program, Alexandria University, Alexandria, Egypt

2. performing the actual insertion or deletion, in constant time
3. rebalancing the tree; this step always takes $O(\log n)$ time.

There are some applications (see [15] for example) in which the position of the node to be inserted or deleted is already known. In this case, the operational complexity depends on the rebalancing time. Based upon B-trees, Guibas et al. [8], Huddleston and Mehlhorn [10], and Overmars [20] demonstrated that their structures achieve constant amortized update time once the position of the key is known.

The idea of decoupling tree rebalancing from other update steps, and delaying the rebalancing, was used to speed up processing of simultaneous requests in main-memory databases, and to allow for concurrency control [5,13,17,18]. Under this scheme, Larsen [12] gave a relaxed red–black tree with constant amortized rebalancing time per update using only standard rotations.

Levcopoulos and Overmars [14] presented a balanced search tree achieving constant worst-case rebalancing time by using a global splitting lemma, which is based on a pebble game, in combination with the bucketing technique of Overmars [20]. In particular, instead of storing single keys in the leaves of their search tree, each leaf stores a bucket of an ordered list of several keys. The buckets in [14] have size $O(\log^2 n)$ each. In accordance, a two-level hierarchy of lists is needed to guarantee $O(\log n)$ query time within the buckets. Additionally, they adopted lazy deletions and relied on global rebuilding [21]. This is a technique that incrementally rebuilds the whole data structure in a background process, naturally consuming extra space and time. Fleischer [7] presented a simpler approach using properties of B-trees and the bucketing technique. He also did not give a direct deletion procedure and relied on global rebuilding. In addition, his proof of correctness is quite involved.

In this paper we give a red–black tree with constant worst-case rebalancing time per operation. Our red–black-tree-based approach is quite similar to Fleischer's implementation in the sense of inserting elements into buckets not in the internal tree and splitting the buckets if they become large. The splitting operation results in adding a new node to the tree, and may cause a violation of the original red–black tree invariants. We will show that these violations can be fixed before the bucket splits again. For deletion in our trees, there is no need for global rebuilding like in [7,14]. In accordance, the structure requires less space and time. The paper is organized as follows: Sect. 2 demonstrates the data structure, Sect. 3 proves the correctness, and Sect. 4 is the conclusion.

2 The data structure

A red–black tree, as described in [9], is a binary search tree with one extra bit per node that represents its color (either red or black). Each node has several attributes: key, left-child pointer, right-child pointer, parent pointer and color bit. If a child of a node does not exist, we make the corresponding child pointer point to a dummy leaf. The *black height* of a node x is defined as the maximum number of black nodes on a path from x to any of its descendant leaves.

The following invariants must be satisfied for a red–black tree:

$(1)_0$ Every node is either red or black.
$(2)_0$ The root is black.
$(3)_0$ Every leaf is black.
$(4)_0$ If a node is red, then both its children are black.
$(5)_0$ The two children of each internal node have the same black height.

In this section we present a slightly relaxed red–black tree that uses the bucketing technique (each leaf is a bucket that is a sorted list of nodes) and must satisfy the following invariants:

(1) Every node is either red or black.
(2) The root is black.
(3) Every leaf is a bucket and is black.
(4) If a node is red and has a red parent, then both its children are black.
(5) The two children of each internal node may differ in their black height by at most one.

A major difference between our trees and the vanilla red–black trees is that the leaves of our trees are buckets. The keys are stored inside the buckets, and each tree node stores a routing value that is smaller than or equal to the keys stored in the buckets in its right subtree and larger than those in its left subtree. A bucket is split once it becomes too large, and two neighbouring buckets are merged once they become too small. Each bucket has a *header* that connects it to a tree node and stores other information about the bucket, including its size. The keys inside a bucket are kept in sorted order in a doubly-linked list. In addition to the key and normal list pointers, each list node stores a pointer to the header of its bucket to be able to identify the bucket in which it resides. For the technical details, how buckets can be split and merged in constant worst-case time, we refer to Sect. 2.4. We relax Invariant $(4)_0$ to allow pairs of consecutive red nodes on any path. This red pair is considered a *double-red conflict* that needs to be fixed incrementally within the upcoming operations. We also relax Invariant $(5)_0$ so that the black heights of the two children for any node differ by at most one. When a black node has a black height that is one less than that of its sibling, to compensate for this difference, we deal with its color as doubly black. This *doubly-black conflict* is to be as well fixed incrementally within the upcoming operations. Our trees can be seen as a stricter version of the relaxed red–black trees of [12], which allow multiple consecutive nodes to be red and allow the difference of black heights between two siblings to be more than one. In contrast to the amortized constant rebalancing time for the relaxed red–black trees of [12], we achieve worst-case constant rebalancing time.

The following lemma bounds the height of our trees.

Lemma 1 *For our red–black trees, the height of a tree that has n nodes is less than* $4.33 \log_2(n + 2)$.

Proof By considering Invariant (5), the black height of any node resembles the height of an AVL tree, which is less than $1.4405 \log_2(n + 2)$. Also, by considering Invariant (4), the number of black nodes on any path is at most one third of the total number of nodes on the path, and thus the height of the tree is less than $3 \times 1.4405 \log_2(n + 2) < 4.33 \log_2(n + 2)$. □

Let $H = \lceil 4.33 \log_2(n + 2) \rceil$ denote an upper bound for the height of a tree of n nodes, in accordance with the above lemma. We maintain the following Invariant on the sizes of our buckets.

(6) The number of keys in each bucket must be in the range $0.5H$–$2H$.

When querying for a key, we follow the search path from the root until reaching a bucket that we scan looking for the key. For insertions (Sect. 2.1) and deletions (Sect. 2.2), we assume that the location of the key to be inserted or deleted is known. A *fixing pointer* is associated with each bucket header and points to a tree node on its search path. A fix-up procedure is called within every updating operation and utilizes the fixing pointer γ_b of the bucket b where the update took place. Let v be the tree node pointed to by γ_b before the fix-up procedure. The following fix-up procedure is executed if v is not the root.

1. If v is red with a red parent, fix v using the *double-red fix-up* procedure (Sect. 2.5).
2. If v is doubly-black, fix v using the *doubly-black fix-up* procedure (Sect. 2.6).
3. Advance γ_b to point to v's parent.

The double-red and the doubly-black fix-up procedures are guaranteed to either eliminate the conflict at node v or propagate it to v's parent or v's grandparent. In accordance, the conflict will be fixed in at most $H + O(1)$ calls to the fix-up procedure for b. We do not stop advancing the fixing pointer to v's parent even if the conflict is eliminated. As we will show in Sect. 3, the validity of the tree (at most two consecutive red nodes on any path and the black height difference between any two sibling nodes is at most one) is maintained.

2.1 Insertion

The insertion algorithm is similar to Fleischer's [7]. One difference is that we split a bucket when the number of its keys is about to overstep that of Invariant (6), and not when its fixing pointer reaches the root. We perform the insertion according to the following algorithm:

1. Insert the new list node at the designated location of the designated bucket b.
2. Call the fix-up procedure for b.
3. If the bucket's size exceeds $2H - 10$:
   (a) If γ_b has not reached the root yet: Repeatedly call the fix-up procedure for b until γ_b reaches the root. (We shall show in Sect. 2.3 that there will be only a constant number of calls).
   (b) Split the bucket into two buckets as shown in Fig. 1. Add their new parent as a tree node in place of the old bucket and color it red. Make the fixing pointers of both new buckets point to the new added node. Call the fix-up procedure for one of the two buckets (to instantaneously dismiss a possible double-red conflict from above the buckets).

Since the bucket's size after splitting is at most H and after merging is less than $H + 6$ (see Sect. 2.2) and the bucket is split when its size exceeds $2H - 10$, it then takes $H - O(1)$ insertions in this bucket since its previous split or merge before it is split next. As we call the fix-up procedure once per insertion, the fixing pointer of the bucket would be at a constant distance from the root when the size of the bucket reaches the splitting threshold. We shall show in Sect. 2.3 the reason for splitting buckets a bit early, at $2H - 10$ instead of $2H$.

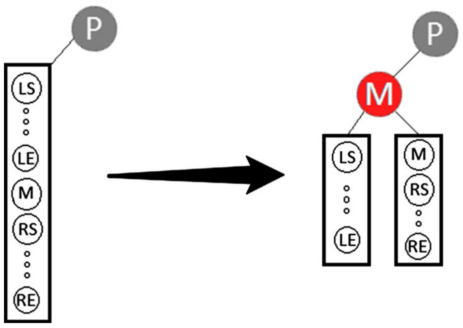

Fig. 1 Splitting a bucket. The value of M is now added to the tree in a red node, while M becomes the first key in the right bucket. As in all other figures, the gray nodes are either black or red. "Left Start" LS and "Right Start" RS are the first keys of the left half and right half respectively, "Left End" LE and "Right End" RE are the last keys of the left half and right half respectively

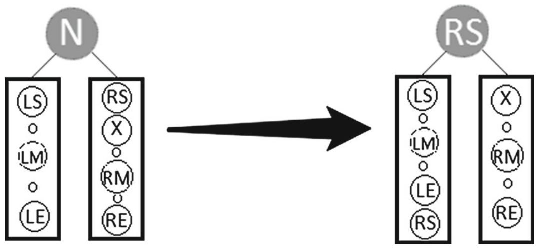

Fig. 2 The left bucket borrows a key from the right bucket by adding the right bucket's first key RS to the end of the left bucket. The parent will have a new value equal to RS. The key X, which follows RS, is now the first key in the right bucket

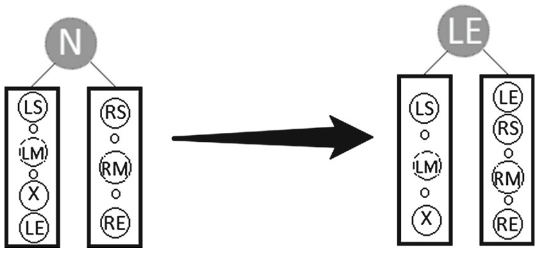

Fig. 3 The right bucket borrows a key from the left bucket by adding the left bucket's last key LE to the start of the right bucket. The parent will have a new value equal to LE. The key X, which precedes LE, is now the last key in the left bucket

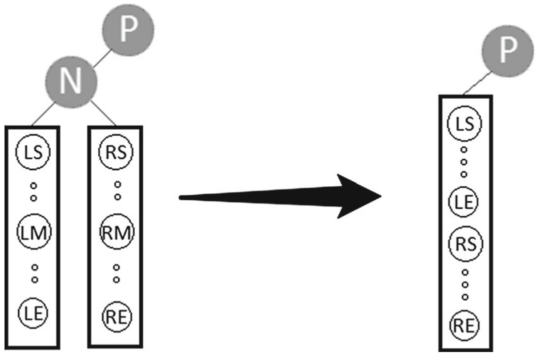

Fig. 4 Merging two buckets. The keys of the right bucket are appended after the tail of the left bucket, and the new merged bucket's parent will be N's parent

2.2 Deletion

Since we assume that all keys are stored in the buckets, deleting a key is thus equivalent to removing a list node from a bucket. We perform the deletion according to the following algorithm:

1. Remove the desired list node from the bucket b.
2. Call the fix-up procedure for b twice.
3. If the bucket's size is less than $0.5H + 3$:
   (a) If γ_b has not reached the root yet: Repeatedly call the fix-up procedure for b until γ_b reaches the root. (We shall show in Sect. 2.3 that there will be only a constant number of calls.)
   (b) If the size of b's sibling bucket b' exceeds $0.5H + 3$: Borrow a node from b' (see Figs. 2 and 3). Call the fix-up procedure for b' twice.
   (c) If the size of b's sibling bucket b' is at most $0.5H + 3$: Merge b and b' into bucket b'' (see Fig. 4). If the deleted common parent was black, mark b'' as doubly-black. Make $\gamma_{b''}$ point to b''.

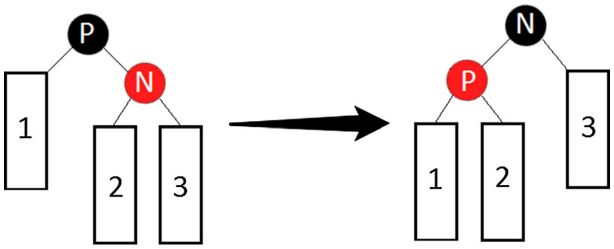

Fig. 5 Bucket 1 has a tree-node sibling. In order to merge with or borrow from bucket 2, we perform a left rotation for P. (Alternatively, if bucket 1 was the right child, we rotate P right)

Since the bucket's size after merging is at least H and after splitting is at least $H - 5$ (see Sect. 2.1) and the bucket will be merged when its size is less than $0.5H + 3$, it takes $0.5H - O(1)$ deletions in this bucket since its previous split or merge before it is merged next. So, we move the bucket's fixing pointer up twice per deletion for the bucket's pointer to be at a constant distance from the root by then. We shall show in Sect. 2.3 the reason for borrowing/merging buckets a bit early, at $0.5H + 3$ instead of $0.5H$.

Note that it is not always the case that a bucket's sibling is a bucket. However, since the bucket's black height is 1, its sibling must have black height 1 before merging or borrowing. This is because the sibling cannot have black height 0 (by Invariant (3) buckets have black height 1), and it cannot have black height 2 (for bucket 1 itself would then be doubly black and this is impossible before merging or borrowing). Also, step 3(b) in the insertion procedure guarantees that we cannot have a red parent and grandparent for a bucket. It follows that Fig. 5 shows the only way for a bucket to have a tree-node sibling and how, with a single rotation, it is guaranteed to have a bucket sibling.

2.3 Global fixing

The fact that H is a function of the number of tree nodes n may cause some bucket sizes to disobey the thresholds, not because of an insertion or deletion in the bucket but because H changes. Recall that H is $\lceil 4.33 \log_2 (n + 2) \rceil$. The value of H increases by four or five when n doubles and decreases by four or five when n halves, and the value of $0.5H$ increases or decreases by at most three. If this issue is left unresolved, Invariant (6) may be violated as H repeatedly increases or decreases.

This problem did not emerge in [7] and [14] as their bucket sizes are only bounded as a function of n from above. Since their trees are only growing in size, because deletions are performed by global rebuilding, a violation of the bound for a bucket takes place only by insertions in this bucket.

Another associated problem we might face as H changes is that some bucket sizes could reach the splitting or merging threshold while their fix-up pointers are not yet at the root. When n doubles, the size of a bucket that was $0.5H + 3$ is now smaller by up to three (may reach $0.5H$). In this case, for those buckets, we are short by fix-ups from up to three deletion operations and need to make up for six additional fix-ups (two per missing deletion). Also, since H increases by at most five, it is possible that the path between their fixing pointers and the root is extended by up to five levels. Therefore, each of those buckets may need up to a total of eleven fix-ups for their pointers to reach the root. Alternatively, when n halves, the size of a bucket that was $2H - 10$ is now more by up to ten (may reach $2H$). In this case, for those buckets, we are short by the effect of ten insertion operations and need to make up for ten additional fix-ups. With time, the number of additional fix-ups needed to reach the root would stack up.

To resolve these issues, we scan over the buckets while performing the insertions and deletions. Within this scan, we call the fix-up procedure a constant number of times per bucket (eleven times if H increases or ten times if H decreases). In addition, we fix by splitting or borrowing/merging the buckets that have sizes more than $2H - 10$ or less than $0.5H + 3$, respectively. Recall that when inserting into a bucket we split this bucket when its size is more than $2H - 10$ (Sect. 2.1), and when deleting from a bucket we borrow/merge this bucket when its size is less than $0.5H + 3$ (Sect. 2.2). Since we scan all the buckets before n doubles or halves, all the bucket sizes will be at least $0.5H$ and at most $2H$, fulfilling Invariant (6). In addition, the fix-ups are synchronized with the operations.

One last piece of non-synchronization might happen. If H changes and the size of a bucket reaches the splitting or merging threshold before doing the aforementioned global fix-ups, the fixing pointer of that bucket would still be within a constant distance from the root. To resolve this issue, within the insertion (Sect. 2.1) and deletion (Sect. 2.2) operations, we perform these constant number of fix-ups until the bucket's fixing pointer is at the root.

2.4 Operations inside the buckets

To perform the operations of the buckets in constant worst-case time, the idea is to distribute the work incrementally inside individual buckets with each update operation. We maintain within the header of each bucket pointers to the first, middle and last list nodes of the bucket. Updating the three pointers when two buckets are merged is straightforward. It is also straightforward to update the middle pointer after insertions and deletions, by moving the pointer once the count of keys on one side of the pointer exceeds the other by two. When a bucket splits, both the resulting buckets will not have the middle pointers correctly set. We adjust the middle pointers incrementally, by initializing a tentative pointer to the beginning for each of the two buckets and moving the pointer forward with every update operation in the bucket, while keeping track of the count of keys on both sides of the pointer. Since the number of operations until the next split is at least as large as the required pointer displacements, it is guaranteed that the middle pointers will be correctly set before then.

When a key is inserted or deleted, we need to access the fixing pointer of the bucket in which this key resides in order to perform the fix-up procedure. For that, each list node needs to refer to the header of the bucket in which it resides. But if we split or merge a bucket, these references may need to be updated for all the list nodes. However, we only have constant time per operation to spare. To resolve this issue, we aim to keep in each bucket's header two copies of the fixing pointer, the left copy is supposed to be referenced by the nodes that precede the bucket's middle pointer and the right copy is supposed to be referenced by the nodes that follow the middle pointer. When a node is inserted it is made to refer to the corresponding copy. Accompanying each updating operation within the bucket, when the middle pointer is moved, we redirect the affected nodes to refer to the corresponding copy. When a bucket is split in two, the left copy is kept with the left bucket and the right copy is kept with the right bucket. In accordance, all the nodes of each of the two split buckets will be referring to one copy. To return back to the two-copies situation, a second copy is created for each of the two buckets, and the nodes preceding the middle pointers are incrementally made to refer to the new copy for both buckets. This is done by incrementally changing the reference of one node with each upcoming update operation in this bucket, guaranteeing that all the changes are finished before the next split or merge operation is due. When two buckets are merged, we will subsequently have four copies instead of two. To return back to the two-copies situation, with each upcoming update operation, we incrementally redirect the references of two of the nodes of the left side to point to one of the first two copies and two more of the right side to refer to one of the other two copies. Once all the nodes of each side point to one copy, we discard the other copy. At the end, we are left with two copies, and this happens before the next split or merge operation is due.

2.5 Double-red fix-up procedure

The double-red fix-up procedure in the vanilla red–black tree aims to handle a red node that has a red parent, as this violates Invariant $(4)_0$. The procedure [6,23] resolves the violation for this node either by $O(1)$ rotations or by a color flip for the parent to become black, so Invariant $(4)_0$ is no longer violated for this node. However, after a color flip, the node's grandparent becomes red and may violate the invariant again. This way another fix is to be performed, and this may be repeated making the procedure run in $O(H)$ time. Instead, we distribute the work over the next operations so that each operation takes constant time. Later, in Sect. 3, we prove that this lazy rebalancing will not invalidate any of the invariants. Our double-red fix-up procedure is almost identical to that of the vanilla red–black tree. For completeness, we give the procedure for the vanilla tree first and then explain the modifications.

We call this procedure for a red node N that has a red parent P, a parent's sibling U, and a grandparent G. In each iteration, this procedure is guaranteed to resolve the conflict or propagate it upward. We only show cases where node P is the left child of G, the other cases are symmetric. The following is the vanilla red–black tree double-red fix-up:

Case 1: If U is red, then color both P and U black and color G red. If G was the root, it stays black. See Fig. 6.
Case 2: If U is black and N is the right child of P, then perform a left rotation at P. Now P is red, its parent N is red and P is the left child of N. We then proceed with Case 3. See Fig. 7.
Case 3: If U is black and N is the left child of P, then perform a right rotation on G, switch the colors of P and G and stop. See Fig. 8.

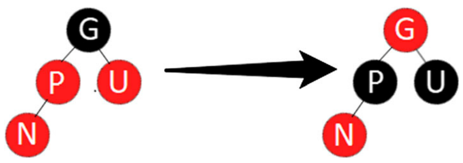

Fig. 6 Double-red fix-up: Case 1

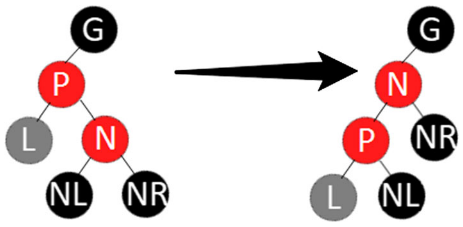

Fig. 7 Double-red fix-up: Case 2

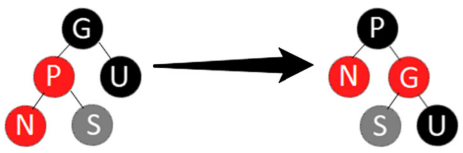

Fig. 8 Double-red fix-up: Case 3

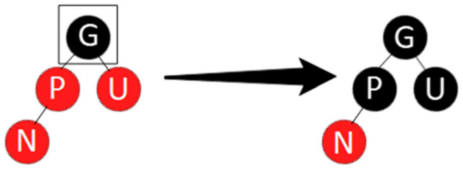

Fig. 9 Double-red fix-up: Case 1.1

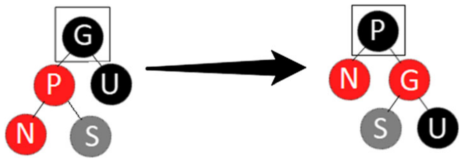

Fig. 10 Double-red fix-up: Case 3.1

It is clear that there are no cases that invalidate the invariants. Moreover, the procedure does not create new doubly-black nodes or two consecutive red nodes except for Case 1, which will create a double-red pair if G's parent was red.

Our modification to this procedure is to perform one iteration and stop, allowing double-red conflicts to exist. In addition, because of the presence of doubly-black nodes, more cases are to be handled. Case 1.1 is similar to Case 1 but treats the situation when G is doubly black. In this case, color both P and U black, keep G black but remove the doubly-black flag (because both its children are now colored black and its black height has increased by one) and stop. See Fig. 9. Case 3.1 is similar to Case 3 but treats the situation when G is doubly black. In this case, perform a right rotation on G, color G red, make P doubly-black and stop. See Fig. 10.

2.6 Doubly-black fix-up procedure

The doubly-black fix-up procedure in the vanilla red–black tree is executed when the deleted node is replaced by its black successor/predecessor. This causes the path that previously contained this successor/predecessor to have one black node less, violating Invariant $(5)_0$. The procedure [6,23] resolves the violation either by $O(1)$ rotations or propagates the doubly-black conflict to the node's parent by color flips. This way, another fix is to be performed and possibly repeated, making the procedure run in $O(H)$ time. In our case, a bucket is colored doubly-black following a merge operation if the parent of the two merged buckets was black. Similar to the double-red fix-up, we distribute the work over the next operations. We show in Sect. 3 that this lazy rebalancing will not invalidate any of the invariants. We give the procedure of the vanilla tree first and then explain the modifications.

We call this procedure for a doubly-black node N that has a parent P and sibling S. In each iteration, this procedure is guaranteed to either eliminate the doubly-black conflict or to propagate it upward. We only show cases where node N is the left child of P, the other cases are symmetric. The following is the vanilla red–black tree doubly-black fix-up:

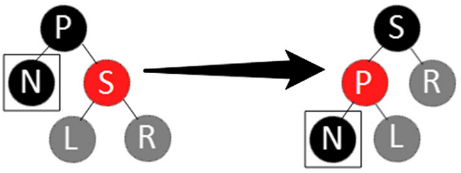

Fig. 11 Doubly-black fix-up: Case 1

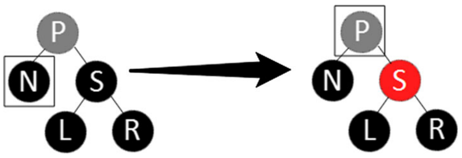

Fig. 12 Doubly-black fix-up: Case 2

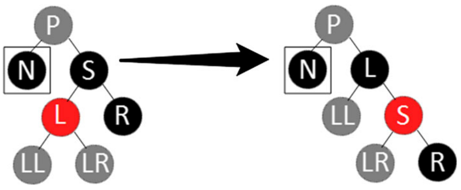

Fig. 13 Doubly-black fix-up: Case 3

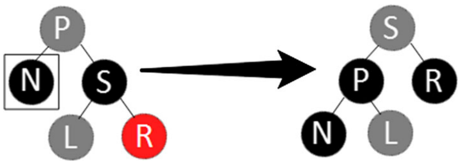

Fig. 14 Doubly-black fix-up: Case 4

Case 1: If S is red and P is black, then perform a left rotation on P, exchange the color of P and S, and proceed to the next cases. See Fig. 11.

Case 2: If S is black and both its children are black, then color S red:

(a) If P is red, color P black and stop.
(b) If P is black and is not the root, set its doubly-black flag. See Fig. 12.

Case 3: If S is black, the right child of S is black and the left child is red, then perform a right rotation on S, exchange its color with its new parent and proceed to Case 4. See Fig. 13.

Case 4: If S is black and its right child is red, perform a left rotation on P, exchange the color of P and S, color the right child of S black, reset N's doubly-black flag and stop. See Fig. 14.

Our modification to this procedure is to perform one iteration and then stop, allowing doubly-black nodes to reside. Since we allow double-red pairs and doubly-black nodes to exist, we need to handle more cases. First, Case 1 would possibly be repeated twice since the new sibling of N after the rotation may also be red. Case 1.1 is a subcase of Case 1 that handles the situation when P is red. A new case is Case 5, where S is doubly black.

Case 1.1: If S is red and P is also red, then perform a left rotation on P but without recoloring, and proceed to the next cases. See Fig. 15.

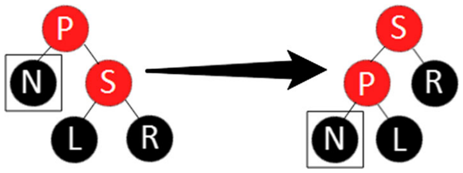

Fig. 15 Doubly-black fix-up: Case 1.1

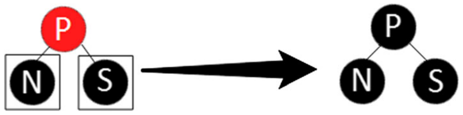

Fig. 16 Doubly-black fix-up: Case 5(a)

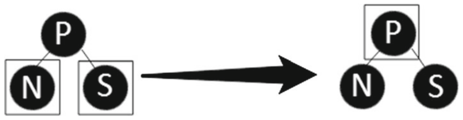

Fig. 17 Doubly-black fix-up: Case 5(b)

Case 5: If S is doubly-black, then remove the doubly-black flag from S and N:

(a) If P is red, color P black and stop. See Fig. 16.
(b) If P is black and is not the root, set its doubly-black flag. See Fig. 17.

The above procedure will either eliminate the doubly-black conflict (Cases $2(a)$, 4 and $5(a)$) or propagate it upward to its parent (Cases $2(b)$ and $5(b)$). Additionally, no new double-red pairs or doubly-black nodes are created. We prove in Sect. 3 that both the double-red and doubly-black fix-up procedures will not invalidate the invariants.

3 Validity of the tree

In this section we prove that the invariants hold. Since we allow double-red and doubly-black conflicts, and as we perform lazy rebalancing, we need to demonstrate that these conflicts would not extend forming more drastic violations. In more details, we need to show that if a node is a black node that has a red parent and a red grandparent, it is impossible for any of its children to propagate a red color to it (by double-red fix-up Case 1), as this will violate Invariant (4). Also, we need to show that for any node that is doubly black none of its children will propagate another doubly-black conflict to it (by doubly-black fix-up Cases 2(b) and 5(b)), as this will violate Invariant (5).

A bucket is said to be *proper* for a node N if its fixing pointer points to one of N's descendants or N itself, otherwise the bucket is non-proper for N. We define the *buffering* for a doubly-black conflict to be a red node and for a double-red conflict to be a pair of consecutive black nodes or a doubly-black node. Our proving patterns for Invariants (4) and (5) are analogous. We will prove that, on the path between any conflict and a descendant conflict of the same type or a descendant non-proper bucket, there must be a buffering for this conflict. The buffering for a conflict will prevent another conflict of the same type to catch the first. It is straightforward to verify that the buffering indeed resolves the conflict that propagates upward once they meet.

Lemma 2 *On any path from a doubly-black node to a descendant doubly-black node or to a descendant non-proper bucket there exists a buffering for the doubly-black conflict, which is a red node.*

Proof We prove our claim by induction on time while applying the fix-up cases one by one. For the base case, the claim holds trivially for an empty tree.

Just before a bucket becomes doubly black (by merging with another bucket), its fixing pointer must have been pointing to the root, and the bucket was non-proper for its ancestors (except for the root, which is never doubly black). Therefore, by the induction hypothesis, there is a red node on the path between the now doubly-black bucket and its first ancestor doubly-black node (if any exists).

For the inductive step by cases, let P be an internal node in the tree. From the doubly-black fix-up procedure, the only way for P to become doubly black is either by Case 2(b) or by Case 5(b). In Case 2(b) (see Fig. 12), P has two children N and S where N is doubly black. If N is a bucket, then it is a proper bucket (its fixing pointer must be pointing to itself, otherwise the doubly-black conflict would have been fixed). If N is an internal node, then by induction, there exists a buffering from P to any doubly-black conflict or non-proper bucket descending from N. Since S becomes red forming a buffering, the claim holds for P with any descendant of S. The claim then holds for P with any of its descendants. In Case 5(b) (see Fig. 17), P has two doubly-black children N and S. If any of them is a bucket, then it is a proper bucket. If any of N or S is an internal node, then by induction, there exists a buffering from P to any doubly-black conflict or non-proper bucket descending from it.

Since some double-red or doubly-black fix-up cases can cause color flips or a rotation of a red node outside a path, what is left to show is that none of these cases will affect our claim. In the doubly-black fix-up Cases 1, 3, 4 (see Figs. 11, 13, 14), we perform a rotation resulting in removing a red node from a certain path. But this is not a problem since node N in those cases was doubly black, thus between it and any ancestor doubly-black node there exists a buffering by induction. Hence, the red node that was removed from the path by the rotation will not invalidate the claim.

In the double-red fix-up Case 1.1 (see Fig. 9), we propagate a red color to a doubly-black node. In the doubly-black fix-up Case 5(a) (see Fig. 16), we propagate a doubly-black conflict to a red node. As a result, in both cases, both the buffering and the doubly-black conflict are eliminated. For all the other cases, if a red node exists on a subpath before the case treatment, it will still reside along the same subpath afterwards. Finally, for all the cases, when a doubly black node propagates upward, it will not override any red ancestor if one exists. Hence the buffering nodes with the ancestors of the black conflict are still surviving. □

Lemma 3 *On any path from a double-red conflict to a descendant double-red conflict or to a descendant non-proper bucket there exists a buffering for the double-red conflict, which is either two consecutive black nodes or a doubly-black node.*

Proof We prove our claim by induction on time while applying the fix-up cases one by one. For the base case, the claim holds trivially for an empty tree.

Just before a bucket splits possibly generating a double-red conflict, its fixing pointer must have been pointing to the root, and the bucket was non-proper for its ancestor nodes (except for the root, which is never red). Therefore, by the induction hypothesis, we have either two consecutive black nodes or a doubly-black node between the created double-red conflict and its ancestor double-red conflict (if any exists). Since the buckets are black, the formed red conflict has no red descendants.

For the inductive step by cases, the only way for a double-red conflict to propagate is by double-red fix-up Case 1. In this case (see Fig. 6), node G is black and node N is red and has a red parent P. After the fix-up, G becomes red and P becomes black. If the parent of G is red, the conflict at N propagates upward to G. By induction, the claim holds for N and hence for G with any descendant of N. Node U is the other child of G and was red. If a child of U is red, it formed a double-red conflict with U. Hence, the claim follows by induction for G with the descendants of this child of U. If a child of U is black, as U becomes black after the fix-up, we get a buffering of two consecutive black nodes between the new red conflict with any descendant red conflict or any of its non-proper buckets.

As some cases can cause color flips or node rotations outside a path, we show next that none of these cases will affect our claim. In the double-red fix-up Case 2 (see Fig. 7), a double-red conflict changes its path. However, this case is followed by the execution of Case 3 that remedies the situation by breaking the conflict. In the double-red fix-up Cases 3 and 3.1 (see Figs. 8, 10), nodes G and U were black forming two consecutive black nodes, and G is turned red. But since nodes N and P were red forming a double-red conflict, thus by induction, there exists another ancestor buffering for the double-red conflict below any ancestor conflict. In the double-red fix-up Case 1.1 (see Fig. 9), node G was doubly-black and after the fix-up G and both its children are colored black. If G was a doubly-black buffering between two double-red conflicts, G and any of its children still form a buffering between the two conflicts. In the doubly-black fix-up Case 2 (see Fig. 12), node S is colored red breaking a two consecutive black pair that forms a possible buffering. If P was black before the fix-up, P will be doubly-black, which is a buffering. If P's parent is black, then P and its parent will form a two consecutive black nodes, which is also a buffering. Alternatively, if both P and its parent were red, a double-red conflict is resolved by the fix-up (since P becomes black), and this permits eliminating the buffering. Also for the doubly-black fix-up Case 3 (see Fig. 13), a black node is colored red breaking a two consecutive black pair that forms a possible buffering. However, this case is followed by Case 4 that remedies the situation by creating a new black pair. In the doubly-black fix-up Cases 4 and 5 (see Figs. 14, 16, 17), a doubly-black node is turned black and its parent becomes black. As above, a possible doubly-black buffering is replaced by two consecutive black nodes forming a replacement buffering. In all cases, the claim still holds. □

4 Conclusion

We have described a relatively simple search tree, a variant of the red–black trees, which supports queries in $O(\log n)$ worst-case time and updates in $O(1)$ worst-case time once the location of the key is known. This tree provides deletions without the global rebuilding technique. We have implemented our structure and verified its correctness and efficiency.[1] It should be an easy exercise to apply our construction to other variants of the red–black trees that have constant amortized rebalancing cost.

References

1. Adel'son-Vel'skii, G.M., Landis, E.M.: An algorithm for the organization of information. In: Proceedings of the USSR Academy of Sciences (1962)
2. Andersson, A.: Balanced search trees made simple. In: Proceedings of the 3rd Workshop on Algorithms and Data Structures, volume 709 of Lecture Notes in Computer Science, pp. 60–71. Springer (1993)
3. Bayer, R.: Symmetric binary B-trees: data structure and maintenance algorithms. Acta Inf. **1**, 290–306 (1972)
4. Bayer, R., McCreight, E.M.: Organization and maintenance of large ordered indices. Acta Inform. **1**, 173–189 (1972)

[1] https://github.com/m-kahla/constant-rebalance-red-black-tree.

5. Boyar, J., Larsen, K.S.: Efficient rebalancing of chromatic search trees. J. Comput. Syst.Sci. **49**(3), 667–682 (1994)
6. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press, Cambridge (1998)
7. Fleischer, R.: A simple balanced search tree with O(1) worst-case update time. Int. J. Found. Comput. Sci. **7**(2), 137–150 (1996)
8. Guibas, L.J., McCreight, E.M., Plass, M.F., Roberts, J.R.: A new representation for linear lists. In: Proceedings of the 9th Annual ACM Symposium on Theory of Computing, pp. 49–60 (1977)
9. Guibas, L.J., Sedgewick, R.: A dichromatic framework for balanced trees. In: Proceedings of the 19th Annual Symposium on Foundations of Computer Science, pp. 8–21 (1978)
10. Huddleston, S., Mehlhorn, K.: A new data structure for representing sorted lists. Acta Inform. **17**, 157–184 (1982)
11. Knuth, D.E.: The Art of Computer Programming: Sorting and Searching, vol. 3, 2nd edn. Addison Wesley Longman Publishing Co., Boston (1998)
12. Larsen, K.S.: Amortized constant relaxed rebalancing using standard rotations. Acta Inform. **35**(10), 859–874 (1998)
13. Larsen, K.S., Ottmann, T., Soisalon-Soininen, E.: Relaxed balance for search trees with local rebalancing. Acta Inform. **37**(10), 743–763 (2001)
14. Levcopoulos, C., Overmars, M.H.: A balanced search tree with O(1) worst-case update time. Acta Inform. **26**(3), 269–277 (1988)
15. Mulmuley, K.: Randomized multidimensional search trees: dynamic sampling (extended abstract). In: Proceedings of the 7th Annual Symposium on Computational Geometry, pp. 121–131 (1991)
16. Nievergelt, J., Reingold, E.M.: Binary search trees of bounded balance. SIAM J. Comput. **2**(1), 33–43 (1973)
17. Nurmi, O., Soisalon-Soininen, E.: Uncoupling updating and rebalancing in chromatic binary search trees. In: Proceedings of the 10th ACM Symposium on Principles of Database Systems, pp. 192–198 (1991)
18. Nurmi, O., Soisalon-Soininen, E.: Chromatic binary search trees. A structure for concurrent rebalancing. Acta Inform. **33**(6), 547–557 (1996)
19. Okasaki, C.: Red–black trees in a functional setting. J. Funct. Program. **9**(4), 471–477 (1999)
20. Overmars, M.H.: A O(1) average time update scheme for balanced search trees. Bull. EATCS **18**, 27–29 (1982)
21. Overmars, M.H.: The Design of Dynamic Data Structures. Lecture Notes in Computer Science, vol. 156. Springer, Berlin (1983)
22. Sedgewick, R.: Left-leaning red–black trees. Technical report, Princeton University (2008)
23. Tarjan, R.E.: Updating a balanced search tree in O(1) rotations. Info. Process. Lett. **16**(5), 253–257 (1983)